

anki_vector.events

Event handler used to make functions subscribe to robot events.

Classes

<code>EventHandler</code> (robot)	Listen for Vector events.
<code>Events</code>	List of events available.

class `anki_vector.events.EventHandler(robot)`

Listen for Vector events.

close()

Stop listening for events. Automatically called by the `anki_vector.robot.Robot` class.

dispatch_event(event_data, event_type)

Dispatches event to event listeners.

dispatch_event_by_name(event_data, event_name)

Dispatches event to event listeners by name.

```
import anki_vector

def event_listener(robot, name, msg):
    print(name) # will print 'my_event'
    print(msg) # will print 'my_event dispatched'

with anki_vector.Robot() as robot:
    robot.events.subscribe_by_name(event_listener, event_name='my_event')
    robot.conn.run_coroutine(robot.events.dispatch_event_by_name('my_event dispatched',
event_name='my_event'))
```

Parameters

- **event_data** – Data to accompany the event.
- **event_name** (`str`) – The name of the event that will result in func being called.

start(connection)

Start listening for events. Automatically called by the `anki_vector.robot.Robot` class.

Parameters

- **connection** (`Connection`) – A reference to the connection from the SDK to the robot.
- **loop** – The loop to run the event task on.

`subscribe(func, event_type, *args, **kwargs)`

Receive a method call when the specified event occurs.

```
import anki_vector
from anki_vector.events import Events
from anki_vector.util import degrees
import threading

said_text = False

def on_robot_observed_face(robot, event_type, event, evt):
    print("Vector sees a face")
    global said_text
    if not said_text:
        said_text = True
        robot.behavior.say_text("I see a face!")
        evt.set()

with anki_vector.Robot(enable_face_detection=True) as robot:

    # If necessary, move Vector's Head and Lift to make it easy to see his face
    robot.behavior.set_head_angle(degrees(45.0))
    robot.behavior.set_lift_height(0.0)

    evt = threading.Event()
    robot.events.subscribe(on_robot_observed_face, Events.robot_observed_face, evt)

    print("----- waiting for face events, press ctrl+c to exit early -----")

    try:
        if not evt.wait(timeout=5):
            print("----- Vector never saw your face! -----")
    except KeyboardInterrupt:
        pass

robot.events.unsubscribe(on_robot_observed_face, Events.robot_observed_face)
```

Parameters

- **func** (`Callable`) – A method implemented in your code that will be called when the event is fired.
- **event_type** (`Events`) – The enum type of the event that will result in func being called.
- **args** – Additional positional arguments to this function will be passed through to the callback in the provided order.
- **kwargs** – Additional keyword arguments to this function will be passed through to the callback.

subscribe_by_name(func, event_name, *args, **kwargs)

Receive a method call when the specified event occurs.

```
import anki_vector

def event_listener(robot, name, msg):
    print(name) # will print 'my_event'
    print(msg) # will print 'my_event dispatched'

with anki_vector.Robot() as robot:
    robot.events.subscribe_by_name(event_listener, event_name='my_event')
    robot.conn.run_coroutine(robot.events.dispatch_event_by_name('my_event dispatched',
event_name='my_event'))
```

Parameters

- **func** (`Callable`) – A method implemented in your code that will be called when the event is fired.
- **event_name** (`str`) – The name of the event that will result in func being called.
- **args** – Additional positional arguments to this function will be passed through to the callback in the provided order.
- **kwargs** – Additional keyword arguments to this function will be passed through to the callback.

unsubscribe(func, event_type)

Unregister a previously subscribed method from an event.

```

import anki_vector
from anki_vector.events import Events
from anki_vector.util import degrees
import threading

said_text = False

def on_robot_observed_face(robot, event_type, event, evt):
    print("Vector sees a face")
    global said_text
    if not said_text:
        said_text = True
        robot.behavior.say_text("I see a face!")
        evt.set()

with anki_vector.Robot(enable_face_detection=True) as robot:

    # If necessary, move Vector's Head and Lift to make it easy to see his face
    robot.behavior.set_head_angle(degrees(45.0))
    robot.behavior.set_lift_height(0.0)

    evt = threading.Event()
    robot.events.subscribe(on_robot_observed_face, Events.robot_observed_face, evt)

    print("----- waiting for face events, press ctrl+c to exit early -----")

    try:
        if not evt.wait(timeout=5):
            print("----- Vector never saw your face! -----")
    except KeyboardInterrupt:
        pass

robot.events.unsubscribe(on_robot_observed_face, Events.robot_observed_face)

```

Parameters

- **func** (`Callable`) – The enum type of the event you no longer wish to be called when an event fires.
- **event_type** (`Events`) – The name of the event for which you no longer want to receive a method call.

`unsubscribe_by_name(func, event_name)`

Unregister a previously subscribed method from an event.

```

import anki_vector

def event_listener(name, msg):
    print(name) # will print 'my_event'
    print(msg) # will print 'my_event dispatched'

with anki_vector.Robot() as robot:
    robot.events.subscribe_by_name(event_listener, event_name='my_event')
    robot.conn.run_coroutine(robot.events.dispatch_event_by_name('my_event dispatched',
event_name='my_event'))

```

Parameters

- **func** (`Callable`) – The method you no longer wish to be called when an event fires.
- **event_name** (`str`) – The name of the event for which you no longer want to receive a method call.

class `anki_vector.events.Events`

List of events available.

`audio_send_mode_changed= 'audio_send_mode_changed'`

Robot event containing changes to the robot's audio stream source data processing mode.

`cube_connection_lost= 'cube_connection_lost'`

Robot event triggered when an object's subscribed connection has been lost.

`nav_map_update= 'nav_map_update'`

Python event containing nav map data.

`new_camera_image= 'new_camera_image'`

Python event containing a processed camera image (`anki_vector.camera.CameraImage` instance)

`new_raw_camera_image= 'new_raw_camera_image'`

Python event containing a raw camera image

`object_appeared= 'object_appeared'`

Python event triggered when an object first receives `robot_observed_object`.

`object_available= 'object_available'`

After the ConnectCube process is started, all available light cubes in range will broadcast an availability message through the Robot.

`object_disappeared= 'object_disappeared'`

Python event triggered when an object has not received a `robot_observed_object` for a specified time.

`object_finished_move= 'object_finished_move'`

Python event triggered in response to `object_stopped_moving` with duration data.

object_moved= 'object_moved'

Robot event triggered when an object starts moving.

object_observed= 'object_observed'

Python event triggered in response to robot_observed_object with sdk metadata.

object_stopped_moving= 'object_stopped_moving'

Robot event triggered when an object stops moving.

object_tapped= 'object_tapped'

Robot event triggered when an object is tapped.

object_up_axis_changed= 'object_up_axis_changed'

Robot event triggered when an object's orientation changed.

robot_observed_face= 'robot_observed_face'

Robot event for when a face is observed by the robot.

robot_observed_object= 'robot_observed_object'

Robot event triggered when an object is observed by the robot.

robot_state= 'robot_state'

Robot event containing changes to the robot's state.

user_intent= 'user_intent'

Robot event triggered after Vector processes voice commands

wake_word= 'wake_word'

Robot event triggered when Vector hears "Hey Vector"